A Tool for Space-Efficient On-the-fly Race Detection*

Yong-Cheol Kim
Dept. of Computer Engr., Int'l Univ. of Korea, Jinju, South Korea, yckim@iuk.ac.kr

Sang-Soo Jun
Dept. of Computer Engr., Kyung Hee Univ., Seoul, South Korea, cernos@khu.ac.kr

Yong-Kee Jun
Dept. of Informatics, Gyeongsang Nat'l Univ., Jinju, South Korea, jun@gnu.ac.kr

Abstract

Detecting races is important for debugging shared-memory parallel programs, because the races result in unintended nondeterministic execution of the programs. Previous on-the-fly techniques to detect races in parallel programs with general inter-thread coordination show serious space overhead which is dependant on the maximum parallelism of the program. This paper presents a tool for space-efficient on-the-fly race detection. This employs a two-pass algorithm which splits a parallel loop with just one event variable into a series of two serializable loops, while preserving the semantics of the original program. The first serializable loop contains all the original dynamic blocks which are executed before the first wait operation in every thread. And, the next serializable loop contains all the original dynamic blocks which are executed after the first wait operation in every thread.

Keywords: *parallel program, inter-thread coordination, on-the-fly race detection, space efficiency, two-pass loop splitting, serializable loop.*

1. Introduction

It is inherently more difficult to write and debug parallel programs [3, 16] than sequential ones. Since the instruction streams in parallel execution can be executed simultaneously, it is difficult to obtain precise information on the execution state and flow of the parallel program. Particularly, shared-memory parallel programs may have a special kind of bugs called data races or access anomalies, in short races.

A race appears when two instructions in different parallel threads perform accesses to a shared variable without proper inter-thread coordination, and when at least one of the accesses is a write to the variable. Detecting races is important for debugging shared-memory parallel programs, because the races result in unintended non-deterministic executions of the program, which makes the debugging more difficult.

To detect races, *on-the-fly detection* [5, 8, 15, 18] instruments the program to be debugged, and monitors an execution of the program. The monitoring process reports races which occur during the monitored execution. This approach can be a complement to *static analysis* [1, 4, 6, 17], because on-the-fly detection can be used to identify realistic races from the potential races reported by static analysis approaches. Although on-the-fly detection may not report as

* This Research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2011-(C1090-1031-0007)).

many races as *post-mortem detection* [2, 12,14], it guarantees to report at least one race for each variable involved in a race.

On-the-fly detection requires less storage space than post-mortem detection, because much of the information collected by the monitoring process can be discarded as an execution progresses. However, this technique still shows serious space overhead for parallel programs with inter-thread coordination. It is because it should maintain information which is as large as the maximum parallelism of the execution, in the worst-case. Our primary interest is to restructure parallel programs to serializable programs for sequential monitoring, while preserving the semantics of original program.

This paper develops our principle [10] suggested previously by proposing a two-pass algorithm which splits a parallel loop with *just one* event variable into a series of two serializable loops, while preserving the semantics of the original program. The first serializable loop contains all the original dynamic blocks which are executed before the first wait operation in every thread. And, the next serializable loop contains all the original dynamic blocks which are executed after the first wait operation in every thread. We assume that a parallel program is already instrumented with additional code including the labeling functions for on-the-fly race detection.

We formulate our problem and motives including the related work in the next section and describe the loop-splitting algorithm of our solution in section 3. We present a tool for space-efficient on-the-fly race detection using the loop-splitting algorithm in section 4. Finally, we conclude our argumentation in the final section.

2. Background

Shared-memory parallel programs may have two parallel constructs [3, 16]: parallel loops and parallel sections. Although here we only use parallel loops such as shown in Figure 1 due to the restricted space of this paper, our technique can also be applied to parallel sections. In an execution of a parallel loop, multiple threads of control are created at a **parallel do** statement and terminated at the corresponding **end parallel do** statement. Parallel constructs may be nested inside parallel constructs. Branches are not allowed from within a parallel constructs to outside the parallel constructs or vice versa.

We assume that all inter-thread coordination is provided via *event* variables. An event variable is always in one of two states: *clear* and *posted*. The initial value of an event variable is always *clear*. The value of an event variable can be set to *posted* with a **post** statement. And it can be tested with a **wait** statement. A **wait** statement suspends execution of the thread which executes it until the specified event variable's value is set to *posted*. On the other hand, the posting thread may proceed immediately. A **clear** statement resets the value of an event variable to *clear*. In this paper, we assume that there is just one event variable for inter-thread coordination and no **clear** statements in the programs.

A *static block* is a structured block of statements which must not contain any wait statement and any external control flow into or out of the static block. Note that a static block may contain post statements. A *dynamic block* is a sequence of instructions from a static block executed by a single thread. Figure 1 shows an example parallel loop considered in this paper, where E is an event variable, B_i is an i-th static block, and C_i is a Boolean condition of the logical IF statement which contains a wait operation.

In an execution of parallel program, two accesses to a shared variable are *conflicting* if at least one of them is a write. If two accesses are conflicting and executed concurrently in two different dynamic blocks, then these accesses constitute a *race*. On-the-fly race detection instruments additional code into the debugged program to monitor and detect races during an execution of the program

```
0: parallel do i = 1, 3
1: <B0>
2: if <C0> then wait E
3: <B1>
4: if <C1> then wait E
5: <B2>
6: end parallel do
```

Fig. 1 A Parallel Loop

In order to detect races on the fly, one needs to determine if a dynamic block's access to a shared variable is logically concurrent and conflicting with any other previous access to the same variable and then results in a race. This requires *detection protocol* to monitor the dynamic blocks that perform accesses to shared variables and to maintain an *access history* for each shared variable during the execution of the program.

Whenever an access to a shared variable occurs, the logical concurrency should be determined between current access and every other previous access in the access history of the variable. The logical concurrency between two accesses is determined from the concurrency information on the two dynamic blocks that perform the accesses, which are called *block labels* [5]. The labels are generated on each creation or termination of dynamic blocks, and may be stored in access histories of shared variables.

The on-the-fly analysis, however, has yet a large space overhead. The required storage space consists of two components in its complexity: one is the space to maintain access histories for all shared variables, and the other is the space to maintain block labels of simultaneously active dynamic blocks. For example, the worst-case space complexity of Task Recycling [5] is $O(VT + T2)$, where V is the number of monitored variables and T is the maximum parallelism of the debugged program.

To eliminate one of these two components, our technique resorts to sequential monitoring of the program execution, because on-the-fly race detection resorts to checking logical concurrency between two blocks regardless of thread scheduling. In this sequential monitoring, the worst-case space complexity of Task Recycling reduces to $O(VT)$, because labels are not required to be maintained for simultaneously active dynamic blocks.

The sequential execution of the program is *undefined* or *deadlocked*, if a wait statement is executed and the corresponding event variable is not already posted. If a program's sequential execution is defined for all input data sets, the program is *serializable*. Our primary interest is to restructure parallel programs into serializable programs for sequential monitoring, preserving the semantics of the original program.

3. The Loop Splitting Algorithm

To make it a parallel loop to be serializable or not to be deadlocked, we should execute all of the post operations before executing any wait operation for the same event variable. The main idea of our technique is to split every original parallel loop with inter-thread coordination into two serializable loops: one loop only for post operations, and the other only for wait operations. The first serializable loop is called the *before-wait loop* which is for all the dynamic blocks executed before the first wait operation in every thread, and the other serializable loop is called the *after-wait loop* which is for all the dynamic blocks executed after the first wait operation in every thread. This technique therefore partitions the set of all dynamic blocks appeared in the execution of original loop into two sets of dynamic blocks

```
0: parallel do i = 1, 3
1: <B0>
2: if <C0> then goto 100
3: <B1>
4: if <C1> then goto 100
5: <B2>
6: 100 continue
7: end parallel do
```

Fig. 2 The Before-Wait Loop

which are defined in sequential execution.

In the before-wait loop, each thread is immediately terminated whenever it arrives at the first wait operation of the original loop. The remaining part of the thread continues to execute in the corresponding after-wait loop. For example, consider the before-wait loop shown in Figure 2 which is constructed directly from the original loop shown in Figure 1 by replacing each **wait** statement with the predefined special **goto** statement and adding a **continue** statement for the **goto**s. This restructuring technique requires every wait statement in the original loop to be in one logical if statement. We resort this to the instrumentation phase that transforms the program to the code in a canonical form. Note that the before-wait loop does not contain any wait statement, but may contain post statements in its static blocks. This means that the before-wait loop does not perform any wait operation, but performs all post operations appeared before the first wait operation in each thread. We construct this before-wait loop in the first pass of our algorithm as follows.

```
read a Term from the original program
while (Term == not null) {
        if (Term-Type == WAIT) {
                replace the wait statement with a goto statement
                replace the event variable with a number
                write the modified Term
        }
        if (Term-Type == LOOP-END) {
                get the event number
                write "<the event number> continue"
        }
        read a Term from the original program
}
```

In the after-wait loop, each thread is created to continue immediately from the first wait operation of the original loop. For example, consider the after-wait loop shown in Figure 3. This figure is constructed directly from the original loop shown in Figure 1 by eliminating the first static block and restructuring each **wait** statement to be executed if it is the first wait operation in its thread. The restructured code sets a Boolean variable *waited* if the wait operation is first in the thread, which makes the thread continue to execute the next static block. Otherwise, it performs the predefined special **goto** statement to execute the corresponding continue statement which is inserted immediately before the next **wait** statement. We construct this after-wait loop in the second pass of our algorithm as follows.

```
0: parallel do i = 1,3                  8: if <C1> then
1: if <C0> then                         9:      wait E
2:      wait E.                         10:    waited = .true
3:      waited = .true.                 11: endif
4: endif                                12: if .not. waited then goto 20
5: if .not. waited then goto 10         13: <B2>
6: <B1>                                 14: 20 continue
7: 10 continue                          15: end parallel do
```

Fig. 3 The After-Wait Loop

```
Read a Term from the original program
while (Term == not null) {
        if (Term-Type == WAIT-IF) {
                set After-wait flag to true
                add "<the wait number> continue"
                write the Term with adding "waited = .true."
                add "if .not. waited then goto <the wait number>"
        }
        if (Term-Type == LOOP-END) {
                determine a statement number
                add "<the statement number> continue"
                write the Term
        }
        if (After-wait == true) {
                write the Term
        }
        read a Term from the original program
}
```

Although the actual total order of restructured program execution is actually different from that of the original program execution, it shows no problem in monitoring the sequential execution for detecting races in the original program. Note that we construct a serializable program by restructuring a parallel program which is already instrumented. The instrumented program is equipped with additional code including the labeling functions for on-the-fly race detection. This approach leads the partitioned loops not to be instrumented for both **end parallel do** statement of the before-wait loop and **parallel do** statement of the after-wait loop. This implies that the monitored sequential execution of restructured program generates the same block labels that are generated in the monitored parallel execution of the original program, preserving the semantics of the original program for sequential monitoring. For on-the-fly race detection of the restructured program, therefore, we can apply any existing scheme for not only block labeling but also detection protocol, which makes our technique general.

4. The Sequential Race Detector

Using the loop splitting algorithm, we have developed a practical tool, called *Seqrace*, for detecting data races in a sequential execution of OpenMP program with event synchronization. This tool has been developed with *wxPython* library to run under a Windows operating system. Figure 4 shows the initial view of the tool. When you click the button 'RUN' in the

view, it opens the main window, shown in Figure 5, and the initial view gets terminated. And when you click 'EXIT', it automatically exits.

Fig.4 The Initial View of Seqrace

The main window has eight buttons. First is the 'About' button that is on the top-left side of the window. It opens a message box, as shown in Figure 6, which provides the information on Seqrace.

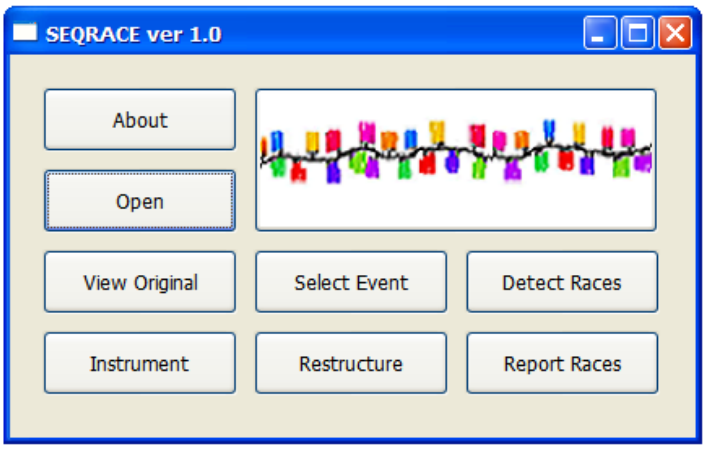


Fig.5 Main Window

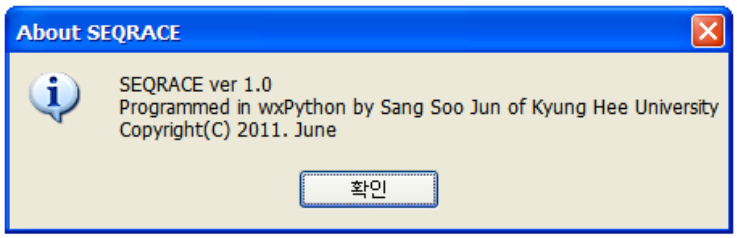


Fig.6 Information on Seqrace

Next is the 'Open' button. When clicked, it opens a directory in the user's computer as shown in Figure 7, so the user can load the program to be debugged. Note that this button only *loads* the program, and nothing else. So, to actually do something with that code, you need to click on the buttons located below.

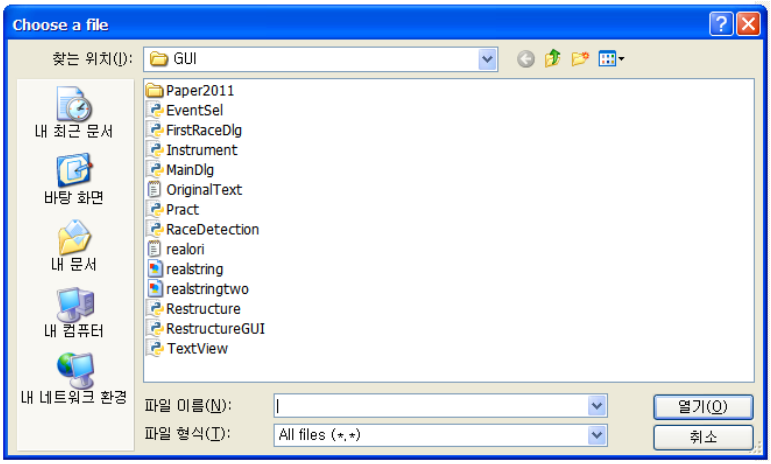
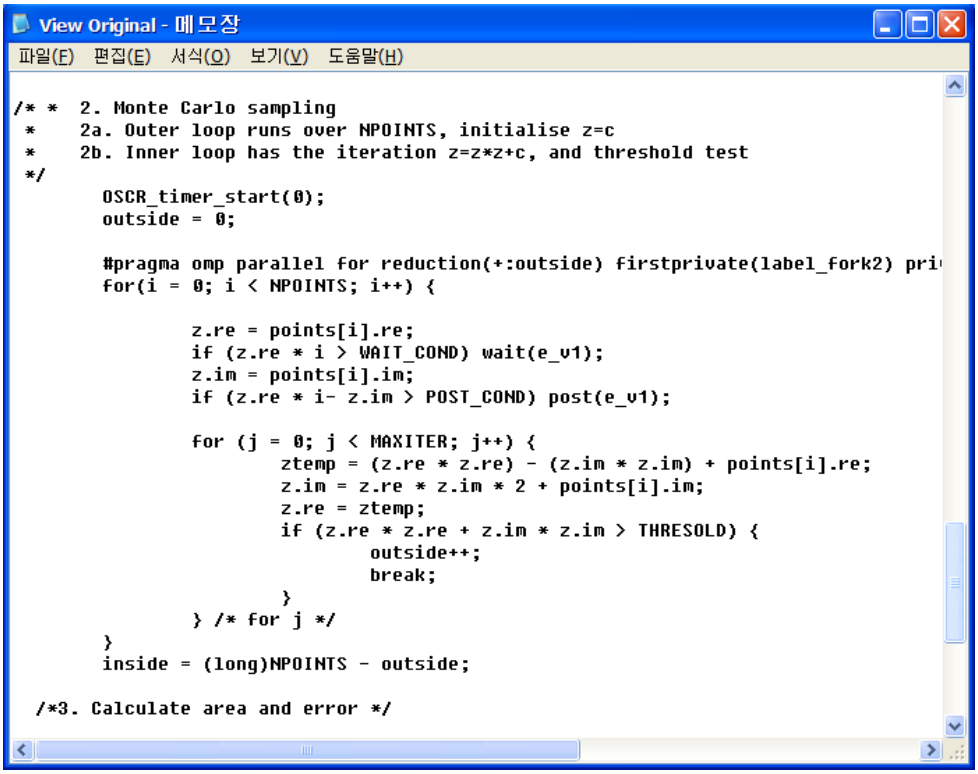


Fig. 7 Load Program

After loading a program to be debugged, we can view the source code of the program by clicking the 'View Original' button. This opens an editor box displaying the source code of the program, as shown in Figure. 8.




```
/* *  2. Monte Carlo sampling
 *    2a. Outer loop runs over NPOINTS, initialise z=c
 *    2b. Inner loop has the iteration z=z*z+c, and threshold test
 */
       OSCR_timer_start(0);
       outside = 0;

       #pragma omp parallel for reduction(+:outside) firstprivate(label_fork2) pri
       for(i = 0; i < NPOINTS; i++) {

               z.re = points[i].re;
               if (z.re * i > WAIT_COND) wait(e_v1);
               z.im = points[i].im;
               if (z.re * i- z.im > POST_COND) post(e_v1);

               for (j = 0; j < MAXITER; j++) {
                       ztemp = (z.re * z.re) - (z.im * z.im) + points[i].re;
                       z.im = z.re * z.im * 2 + points[i].im;
                       z.re = ztemp;
                       if (z.re * z.re + z.im * z.im > THRESOLD) {
                               outside++;
                               break;
                       }
               } /* for j */
       }
       inside = (long)NPOINTS - outside;

 /*3. Calculate area and error */
```

Fig. 8 View Original Program

To instrument this code, click on the 'Instrument' button. When clicked, it will instrument the loaded original program. And a dialog shown in Figure 9 will pop up, asking whether if you want to view the instrumented code or not. When you click 'No', the dialog will disappear. But when you click 'Yes', it will return an editor box displaying the instrumented code as shown in Figure 10.

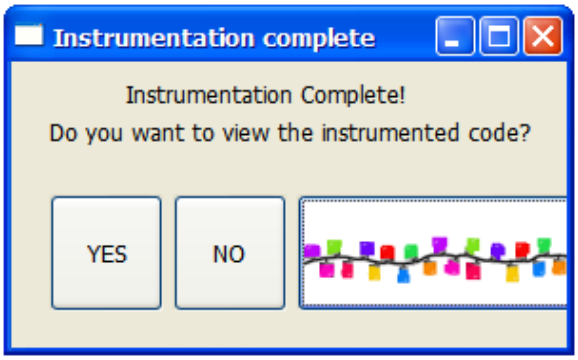


Fig.9 Instrumentation Dialog

View Instrument - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
        outside = 0;

        #pragma omp parallel for reduction(+:outside) firstprivate(label_fork2) privat
        for(i = 0; i < NPOINTS; i++) {
                label_fork2 = (Label_CNR*)CNR_Fork(label_fork1, i, 1, 0, NPOINTS, 1, 1

                z.re = points[i].re;
                Mell_R_Checker(label_fork2, SV, 120);
                label_wait2 = (Label_CNR*)CNR_Wait(label_wait1, i, 1, 0);
                if (z.re * i > WAIT_COND) wait(e_v1);
                z.im = points[i].im;
                Mell_R_Checker(label_fork2, SV, 121);
                label_post2 = (Label_CNR*)CNR_Post(label_post1, i, 1, 0);
                if (z.re * i- z.im > POST_COND) post(e_v1);

                for (j = 0; j < MAXITER; j++) {
                        ztemp = (z.re * z.re) - (z.im * z.im) + points[i].re;
                        z.im = z.re * z.im * 2 + points[i].im;
                        z.re = ztemp;
                        if (z.re * z.re + z.im * z.im > THRESOLD) {
                                outside++; break;
                        }
                } /* for j */
        }
        NR_Join(label_fork1, label_fork1, 130);

        inside = (long)NPOINTS - outside;

        area = 2.0 * (2.5 * 1.125) * inside / NPOINTS;
        error = area / sqrt(NPOINTS);
```

Fig. 10 View Instrumented Program

To restructure the code, the user first needs to select an event variable used in the program by clicking the 'Select Event' button. This will show a dialog with a list box containing all event variables used in the program, as shown in Figure 11. Click on one of the event variables displayed in the list box, and click the 'OK' button.

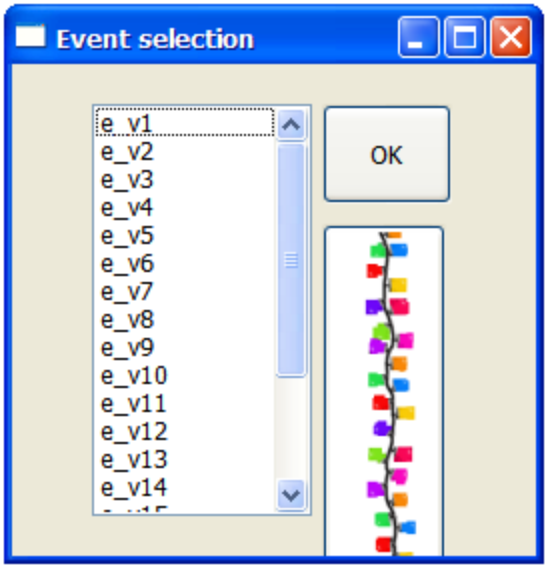


Fig. 11 Select Event Dialog

And then, the user can click on the 'Restructure' button located at the right side of the 'Instrument' button. If you click this, the instrumented code will be restructured, and a dialog shown in Figure 12 will pop up, asking whether if you want to view the restructured code. Note that an error will occur if you click on the 'Restructure' button, before you click the 'Instrument' button. It is because that instrumentation must be done before restructuring. When you click 'No', the dialog will be terminated. But when you click 'Yes', an editor box displaying the restructured code will appear, as shown in Figure 13.

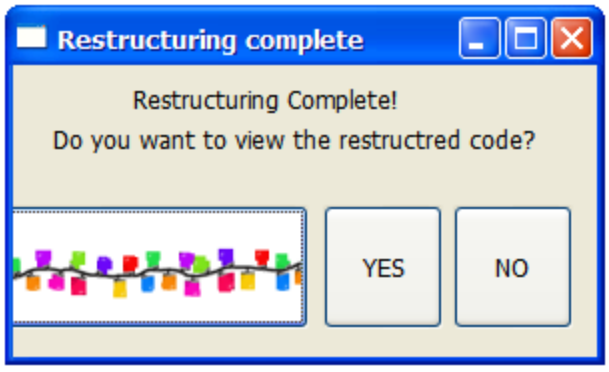


Fig. 12 Restructure Dialog

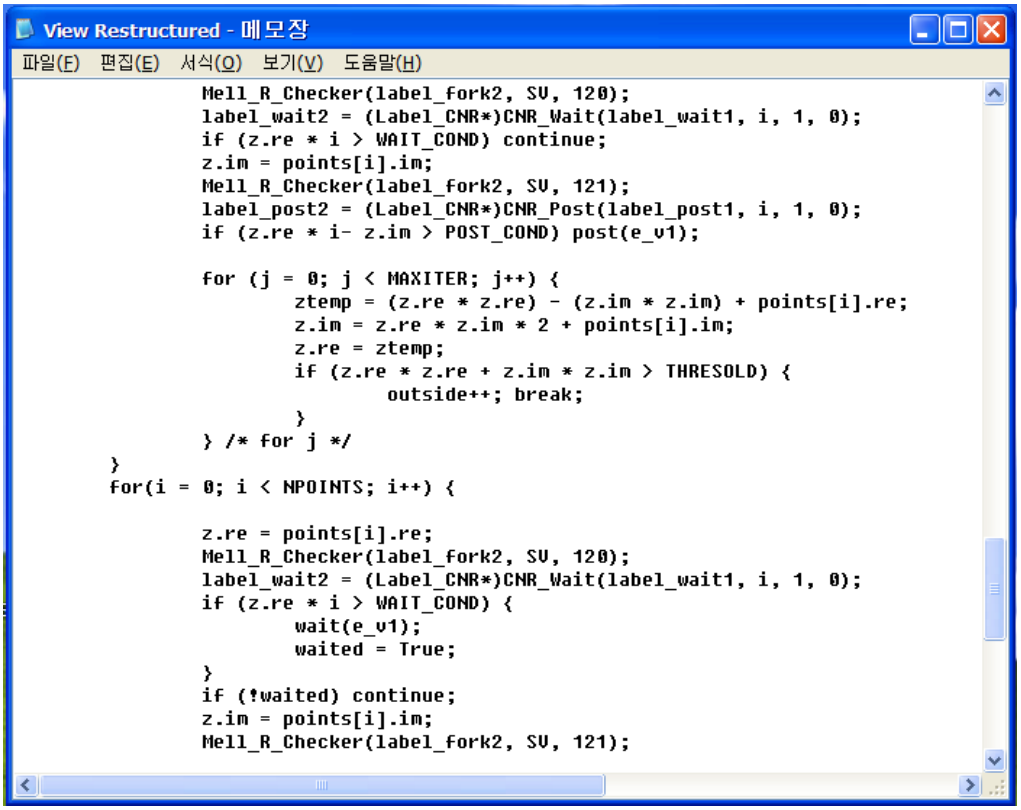


Fig.13 View Restructured Program

If you want to actually detect data races, just click the button with the label 'Detect Races'. When you click this button it will start compiling of the restructured program and run the resulted executable program.

And finally, to check if there exist data races occurred in the program, click the 'Report Races' button. A dialog shown in Figure 14 will be displayed. If you want to detect races on the original program, check the first radio box. Or if you want to detect races on the restructured program, click the second one, and then click the 'OK' button. Figure 15 shows the message boxes to report that the data races have not been detected. The left box reports it in case of the original program, and the right box in case of the restructured program. But, if data races exist, an editor box will appear, as shown in Figure 16.

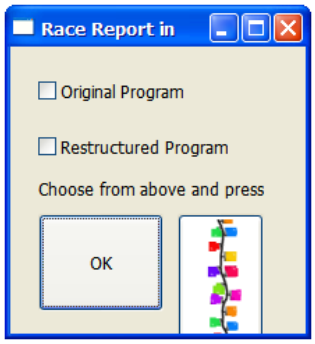


Fig. 14 Report Race Dialog

Fig. 15 No Race Reported

Fig.16 A Race Report

5. Conclusion

On-the-fly techniques used to detect races still shows serious space overhead for parallel programs with inter-thread coordination. It is because it should maintain in formation which is as large as the maximum parallelism of the execution, in the worst-case. This paper proposes a two-pass algorithm which restructures parallel programs with just one event variable to be serializable for sequential monitoring by splitting a parallel loop into a series of two serializable loops, while still preserving the semantics of the original program. A previous work [10] presents a principle to extend this technique for the programs with multiple event variables.

This technique especially allows us to use a two-pass on-the-fly algorithm [15] also for detecting the first races [7, 8, 15] in shared-memory parallel programs with general inter-thread coordination. It is because that this on-the-fly algorithm works for a special class of parallel programs which may have the ordered synchronization. In this kind of programs any pair of the corresponding coordination points is executed on ordered sequence including sequential execution of these points. The first races to occur are important in debugging, because the removal of such races can make other races disappear.

References

[1] Callahan, D., Kennedy, K., Subhlok, J.: Analysis of Event Synchronization in a Parallel Programming Tool. In: 2nd Symposium on Principles and Practice of Parallel Programming. pp. 21-30. ACM. New York (1990)

[2] Chen, F., Serbanuta, T. F., Rosu, G.: jPredictor: A Predictive Runtime Analysis Tool for Java. In: 30th International Conference on Software Engineering (ICSE). pp. 221-230. ACM. New York, USA (2008)

[3] Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. In:

Computational Science and Engineering, 5(1): 46-55. IEEE. Alamitos, USA (1998)

[4] Dabrowski, F., Pichardie, D.: A Certi¯ed Data Race Analysis for a Java-like Language. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOL), Munich, Germany. Lecture Notes in Computer Science (LNCS), 5674: 212-227, Springer-Verlag, Heidelberg, Germany (2009)

[5] Dinning, A., Schonberg, E.: An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In: 2nd Symposium on Principles and Practice of Parallel Programming. pp. 1-10. ACM. New York, USA (1990)

[6] Grunwald, D., Srinivasan, H.: Efficient Computation of Precedence Information in Parallel Programs. In: 6th Workshop on Languages and Compilers for Parallel Computing. pp. 602-616. Springer-Verlag, Heidelberg, Germany (1993)

[7] Ha, K., Jun, Y., Yoo, K.: Efficient On-the-fly Detection of First Races in Nested Parallel Programs. In: Workshop on State-of-the-Art in Scientific Computing (PARA). pp. 75-84, Copenhagen, Denmark, Springer-Verlag, Heidelberg, Germany (2004)

[8] Jun, Y., McDowell, C. E.: On-the-fly Detection of the First Races in Programs with Nested Parallelism. In: 2nd International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1549-1560. CSREA. USA (1996)

[9] Kim, D., Jun, Y.: An Effective Tool for Debugging Races in Parallel Programs. In: 3rd International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 117-126. CSREA. USA (1997)

[10] Kim, Y., Jun, Y.: Restructuring Parallel Programs for On-the-fly Race Detection. In: 5th International Conference Parallel Computing Technologies, pp. 446-452, Russian Academy of Science. Russia (1999)

[11] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. In: Communications of the ACM, 21(7), pp. 558-565. ACM. New York (1978)

[12] Marino, D., Musuvathi, M., Narayanasamy, S.: LiteRace: Effective Sampling for Lightweight Data-Race Detection. In: ACM Sigplan Confernece on Programming Language Design and Implementation (PLDI). Dublin, Ireland. ACM. New York, USA (2009)

[13] Netzer, R. H. B., Ghosh, S.: Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization. In: International Conference on Parallel Processing, pp. 242-246. Pennsylvannia State University, USA (1992)

[14] Netzer, R.H.B., Miller, B.P.: Improving the Accuracy of Data Race Detection. In: 3rd Symposium on Principles and Practice of Parallel Programming. pp. 133-144. ACM. New York, USA (1991)

[15] Park, H., Jun, Y.: Detecting the First Races in Parallel Programs with Ordered Synchronization. In: 6th International Conference on Parallel and Distributed Systems, pp. 201-208. IEEE. Alamitos, USA (1998)

[16] Parallel Computing Forum: PCF Parallel Fortran Extensions. Fortran Forum, 10(3). ACM. New York (1991)

[17] Qadeer, S., Wu, D.: KISS: Keep It Simple and Sequential. In: Sigplan Conference on Programming Language Design and Implementation (PLDI). Washington D.C. Sigplan Notices, 39(6): 14-24, ACM, New York, USA (2004)

[18] Sack, P., Bliss, B. E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and Efficient Filtering for the Intel Thread Checker Race Detector, In: 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID). pp. 34-41. San Jose, California. ACM, New York (2006)

Authors

Young-Cheol Kim received his PhD degree in Computer Science from Gyeongsang National University in 2000. He is now an associate professor in Department of Computer Engineering, International University of Korea (IUK). His research interests include parallel and distributed computing, embedded systems, and system software.

Sang-Soo Jun is an undergraduate student, Department of Computer Engineering, Kyung Hee University, Seoul, South Korea. His research interests include embedded software, bioinformatics, image processing, and computer vision.

Yong-Kee Jun received his PhD degree in Computer Science from Seoul National University. He was a research associate in Computer Science at Univ. of California, Santa Cruz, and a research staff in Electronics and Telecommunications Research Institute (ETRI). Dr. Jun is now a full professor of Gyeongsang National University (GNU) where he served as the founding professor of Dept. of Informatics. He is now the director of GNU Embedded Software Center for Avionics which is a national IT Research Center of South Korea. His research interests include parallel/distributed computing, embedded systems, and systems software. His research has focused primarily on reliability of concurrent software.